Programming Language Concepts

For Multiprocessors

by

Harry F. Jordan

Computer Systems Design Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309-0425

CSDG 87-4
September 1987

Programming Language Concepts for Multiprocessors†

Harry F. Jordan

Abstract

It is currently possible to build multiprocessor systems which will support the tightly coupled activity of hundreds to thousands of different instruction streams, or processes. This can be done by coupling many monoprocessors, or a smaller number of pipelined multiprocessors, through a high concurrency switching network. The switching network may couple processors to memory modules, resulting in a shared memory multiprocessor system, or it may couple processor/memory pairs, resulting in a distributed memory system.

The need to direct the activity of very many processes simultaneously places qualitatively different demands on a programming language than the direction of a single process. In spite of the different requirements, most languages for multiprocessors have been simple extensions of conventional, single stream programming languages. The extensions are often implemented by way of subroutine calls and have little impact on the basic structure of the language. This paper attempts to examine the underlying conceptual structure of parallel languages for large scale multiprocessors on the basis of an existing language for shared memory multiprocessors, known as the Force, and to extend the concepts in this language to distributed memory systems.


Introduction

Early multiprocessors were capable of supporting only a few instruction streams executing simultaneously. In this environment, it is not hard to write separate code for each instruction stream. Even if a common body of code serves to guide several processes, the programmer can be aware of each process as a distinct entity and use conditional branches in the code to cause different processes to do distinct things. When the number of processes becomes a significant fraction of a hundred, it is no longer possible to treat processes individually. A single body of code must be able to direct the activity of many processes simultaneously. This is often based on assigning different portions of a large data structure to different processes and specifying that each processor do the same operations on its portion of the data.

There are two major types of multiprocessor, distinguished by the way in which results produced by one process are communicated to another. Results are either placed in a shared memory to be accessed by another processor, or they are explicitly sent to the process needing them, which is assumed to be running on a processor sharing no memory with the one producing them. On a shared memory system, it is not too hard to produce programming language concepts which relieve the programmer of having to explicitly direct the activity of each of many processes. The uniform sharing of memory helps in writing programs which suppress the individuality of processes. A language for writing such programs on shared memory multiprocessors, called the Force[1], has been implemented on five different shared memory multiprocessors. It classifies variables as either private to a single process or uniformly shared among all processes. Commands are provided to spread the work on different portions of a shared data structure over many processes.

This paper discusses the possibility of extending the ideas embodied in the Force programming language to distributed memory multiprocessors. The key problem is to prevent the need to explicitly communicate values between processors from forcing the programmer to manage each process explicitly. The approach taken is to define several types of variables to be supported by a distributed memory multiprocessor language, and to develop methods for updating versions or parts of these variables in the memories of the individual processors. Variables which are accessible to many processors require many memory cells for their representation, and the updating of such variables requires global communication patterns involving all processors. Such global communications are expensive in resource utilization, but may be the only way to successfully manage a really large number of tightly coupled processors. By providing support for uniform read and update access to conceptually shared variables in a distributed memory multiprocessor, it is possible to simplify the problem of mapping a computation onto a specific system. The mapping problem is reduced to load balancing and the reduction of interprocessor communication.
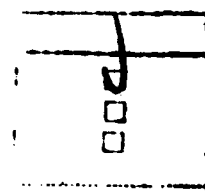
Programming Systems with a Large Number of Processes

The first issue in multiprocessor algorithm structure is how the set of all operations to be performed is divided into sets of operations to be done by each process. One way is to decompose the problem data into items which can be processed in parallel by a single control structure which will become a single body of code executed by all processes. A parameterized reference mechanism (indexing) causes the code to refer to different data items when executed by different processes. In algorithms with a strong Euclidean space connection, such as partial differential equation solution, data decomposition is often called domain decomposition. In addition to decomposition of an algorithm on the basis of data, it is also possible to partition the operations into classes or functions which can be applied in parallel. A simple division in the taxonomy of multiprocessor algorithms may thus be made on the basis of data decomposition versus functional decomposition. Functionally parallel algorithms can perhaps be further classified by the way in which the parallel functions share data. For example, the macro-pipelining structure is well known from the use of coroutines [2] even on uniprocessors.

Although work may be distributed on the basis of data decomposition or on the basis of a functional decomposition, data decomposition is essential in the case of a large number of processes. Since the number of independent functions is limited, some data decomposition of an algorithm is surely necessary when there are many processes. Thus data decomposition is more fundamental to large scale multiprocessor programming than functional decomposition. Even a high level decomposition into a few parallel functions would involve a large amount of data based parallel decomposition within each function.

In addition to biasing parallel algorithm structure in favor of data decomposition, the existence of a large number of processes has other qualitative effects. In the interests of fault tolerance and portability, the correctness of a program should not depend on the number of processes executing it. Only the performance should vary if a few more or less processes are used. Also, coordination of the work between various processes can be done in a "democratic" or "master-slave" fashion. When the number of processes is very large, the master tends to become a serious bottleneck in terms of performance. The democratic, or distributed, coordination of work is thus favored in this environment. Good examples of the importance of such bottlenecks appear in the discussions of shared queue access in the NYU Ultracomputer [3], and in the technique of self-scheduling used by many applications on the HEP [4] multiprocessor. Another qualitative effect is on process naming. Simple identifiers are totally inadequate for naming many processes. Computable names, at least, are essential. Process names with an associated integer index is one way of handling this, but it is better if process individuality can be suppressed entirely when there are hundreds of processes.

The effects described above appear in somewhat different forms in shared and distributed memory multiprocessors. In a shared memory system, data decomposition really amounts to the distribution of the *operations* on

different parts of a data structure over the processes. The data itself is not really decomposed, but resides in the equally accessible shared memory. Of major importance in this case is the synchronization of processes which access the same shared memory cells. This really amounts to the producer of a data item informing its subsequent consumer that the item is available. Synchronization mechanisms involve the conditional delay of consumers. The many process name management problem implies that synchronizations should either refer to processes in a non-specific manner, e.g. mutual exclusion, wait for all, etc., or be based on the data item being shared.

In a distributed memory system, the data is actually distributed among the private memories of the processes using it. Operations on the same data item must either be carried out by the same processor, or the item must be explicitly communicated using a *send* in one process and a *receive* in another. Conditional delay of the consumer is accomplished by a possible wait in the *receive* operation. Thus both data and operations must be distributed over the processors of the system. An issue often discussed in distributed memory systems is the inequality of communication paths between different processor pairs. Except for very regular algorithms, it is quite difficult for a programmer to take explicit account of this in coding for a very large number of processors. Such multiprocessors usually have a rich communications network with a maximum distance no more than logarithmic in the number of processors. In addition, hardware and operating system support for message routing, as well as message header and buffering overhead, tends to minimize the effects of unequal path length. We will take the attitude in what follows that it is not feasible to provide language features which depend on differential path length for communication among a very large number of processors.

A Shared Memory Language - the Force

The Force is a programming language for shared memory multiprocessors which is based on Fortran and addresses the problem of programming for very many processes. A Force program consists of a single body of code executed by an arbitrary number of identical processes. Although the number of processes is not specified in the program, it is assumed to be fixed at the beginning of execution time, and remains fixed until program completion. A Force program may be executed by only one process, and this feature often aids in debugging errors in process synchronization. The language provides execution time access to the number of processes. This number is usually included in program output, and could be used to dynamically select a more efficient algorithm within some range of its value.

The management of individual processes by name is completely suppressed. An integer process index is available at execution time, but it is never necessary for the user to access it explicitly, except possibly for debugging purposes. Parallel primitive operations of the language use this index, but in a manner transparent to the programmer. For example, operations to spread the execution of a parallel loop (DOALL) over all processes may automatically assign loop index values to different processes based on their

process index. It is also possible to use self scheduling to distribute values of the index to processes, in which case not even the implementation needs to use the process index.

The treatment of the parallelism characteristics of variables is fundamental to the Force programming philosophy. The Force allows variables which are either strictly private to each process or uniformly shared among all processes. Thus for a *private* variable P the single name will correspond to a different memory cell (and value) for each process, while a *shared* variable S will correspond to one shared memory location, accessible to all processes. *Private* variables might reside in private memories if the system includes them, or they might simply be separate shared memory cells with a different address for each process. The above parallelism storage class distinction is independent of any storage class specified by the underlying sequential language, such as the Fortran distinction between *local* and *common* variables.

The synchronization required to control access to variables in shared memory is also kept independent of process identity. All synchronizations are "generic" in the sense that they do not designate specific processes. Examples of generic synchronizations are critical sections, which specify that only one process at a time may execute a body of code, and the barrier, which specifies that no process may proceed past a given point until all have arrived. Producer/consumer synchronization is also supported. Delays are imposed by this synchronization on the basis of the implicit state, existent or nonexistent, of a named variable, but the names of the processes accessing the variable are irrelevant. The limitation to a single program for all processes, the uniformity of access to shared variables, the suppression of the process index, and the generic nature of synchronizations all combine to eliminate any underlying system topology from the model of parallel computation supported by the Force language.

To give an idea of the Force language, a simple matrix multiply program is shown in Figure 1. Note that parallel execution is the normal mode. Statements will be executed by all processes unless controlled by a specific parallel construct. An extended version of the barrier synchronization allows inclusion of inherently sequential operations, such as I/O. The semantics are that all processes must arrive at the *Barrier* statement, then one arbitrary process executes the code between *Barrier* and *EndBarrier*, after which all processes proceed with the statement following *EndBarrier*. Assignment statements executed by all processes should either have a *private* variable on the left hand side, or perhaps a *shared* array which is indexed by a *private* variable. Assignment statements inside a barrier should be to *shared* variables so that values are available to all processes outside the barrier.

Concepts for Distributed Memory

The parallel programming methodology represented by the Force has been influenced primarily by two things. The first is the desire to write

5

```
         Force MATMP of NP ident ME
         Shared REAL A(200,200), B(200,200), C(200,200)
         Shared INTEGER M, LDIM
         Private INTEGER I, J, IRES
         Private REAL SUM
         End declarations
         Barrier
C            The order of the matrices - M
         READ (5, 50) M
50          FORMAT(I4)
C            Echo the input with number of processes.
         WRITE (6, 75) M, NP
75          FORMAT(' Order',I4,' matrices using',I4,' processes.')
         End Barrier
C        Set up the matrices to be multiplied.
         Presched DO  10 I = 1, M
            DO  5 J = 1, M
               A(I. J) = 1./3.
               B(I, J) = 3.
5           CONTINUE
10       End Presched DO
         Barrier
         End Barrier
         Selfsched DO  300 I = 1, M
C            Produce all of row I of the C matrix.
            DO  200 J = 1, M
               SUM = 0.0
               DO  100 K = 1, M
                  SUM = SUM + A(I,K)*B(K,J)
100            CONTINUE
               C(I,J) = SUM
200         CONTINUE
300      End Selfsched DO
         Barrier
C            Write the results.
            DO  20 I = 1, M
               WRITE (8, 500) (C(I, J1), J1 = 1,M)
20          CONTINUE
500      FORMAT(6E13.6)
         End Barrier
         Join
         END
```

Figure 1: A Matrix Multiply Force Program.

programs for very many processes, and the second is the underlying shared
memory model of computation. To move the methodology into the

distributed memory environment, we must consider what features are inherent to the many process situation. The bias toward data decomposition is still present. We again bypass the problem of a high level decomposition into a few functions, and treat the data decomposition environment which must exist within each function in order to employ many processes. We therefore consider the case in which the same code is loaded into the program memories of all processors. For simplicity, we assume a static number of processors, all of which start together and are available until the end of the program. Independence of the program from the number of processors is still important. Even if we agreed on some restricted interconnection topology, such as a full hypercube, it would still not be desirable to write a program which could run only on a hypercube of a given dimension.

The way in which a parallel language supports references to variables underlies the computational model it represents, and in distributed memory systems, there are more identifiable parallel storage classes than in a shared memory multiprocessor. If we assume that the *private* versus *shared* distinction has meaning at the parallel algorithm level, we must see how this is reflected in a program for a non-specific distributed memory multiprocessor. We assume only that all memory is divided among the processors, that any cell is accessible to only one of them, and that communication among the processors is entirely by means of message passing. *Private* variables can be handled immediately. They are the same as in the shared memory case; a single name refers to a distinct cell in each processor's memory. A concept useful later in describing structured variables is that of a *unique* variable, represented by one memory cell in the memory of a single processor. A *unique* simple variable would have algorithmic meaning if one processor had unique capabilities, as in the case of a special I/O processor.

Data considered shared in an abstract parallel algorithm are treated in several different ways in actual distributed memory multiprocessor programs. The first case is that of a variable with a single value available to all processors. It must be represented by one cell in each processor's memory. To keep the values of all the representatives the same, any update must be performed cooperatively by all the processors. We will call such variables *cooperative update* shared variables. The special case of a read-only shared variable fits well here. The alternative of having a single processor "own" a shared variable and supply its value to others upon request seems untenable in the many processor case as a result of the performance penalty imposed by the sequential bottleneck.

Another type of "shared" variable is sometimes used in distributed memory multiprocessor programs. It is often more efficient to make a value available to many processors by carrying out redundant computations rather than by communicating its value. The simplest example is that of a loop index which should step through the same range in each processor. Its value should surely be incremented independently by each processor rather than being broadcast from a single processor which calculates it. We will call such variables *replicated*. Note that, although a *replicated* variable will take on the same sequence of values in every processor, a given update to the

7

representative in one processor may occur at a somewhat different time than the corresponding update in a different processor, since processor speeds may vary. This is in contrast to changes to a *cooperative update* variable, which are done simultaneously by all processes cooperating by way of message passing.

Structured variables, vectors, arrays, etc., bring up a different aspect of the parallel storage class of variables. Each component of a structure has one of the above classes, but the structure as a whole may present a different aspect. *Private* structures are straightforward, and the simple case of a *shared* structure, where all elements of the structure are *shared* is useful. Large arrays, however, are probably distributed in such a way that only a fraction of the structure is stored in any one processor. An array distributed by rows, for example, might be considered to have each element of class *unique*. The indexed array name A(I,) could be considered to have parallel class *unique*, but the entire structure A is known to more than one processor, so the array as a whole is in some sense shared. A parallel loop over the row index involving A in the loop body should, ideally, distribute the work over the processors so that each processor handles index values for its own rows. Of course, this is only possible when the loop does not specify combining elements from different rows in an arithmetic operation. We will name the storage class of such variables *fragmented*. Various fragmentation patterns are possible for a structure, and a language should attempt to suppress or simplify these details as much as possible. At the minimum, there should be automatic translation of algorithm level indexing into an address in the correct processor's memory. Note that the specific fragmentation pattern will fix the way in which parallel operations on the structure are allocated to different processors.

Figure 2 summarizes the parallel storage classes identified for distributed memory multiprocessors. Each class has its use in a distributed memory multiprocessor program. *Private* variables are commonly used for temporaries and intermediate results. Major data structures of an algorithm are usually shared in some way. *Cooperative update* variables require extensive communications for any write and are best used for algorithm parameters which change infrequently. *Replicated* variables require redundant computation and should be used only if updates are computationally simple. *Fragmented*

Simple Variables	Structured Variables
Unique	Unique
Private	Private
Cooperative update	Cooperative update
Replicated	Replicated
	Fragmented

Figure 2: Parallel Storage Classes in Distributed Memory

structures do not require an amount of storage which grows with the number of processors. Although they limit access, they are essential for handling very large structures in a distributed memory system.

When a *fragmented* variable has a structure associated with some Euclidean space, and the fragmentation pattern associates connected subregions with each processor, the term *domain decomposition* is often used. In many algorithms, only processors containing data for spatially adjacent subregions need to communicate values. This leads to the idea of processors being *neighbors*, and to nearest neighbor communication patterns. Note that this idea is based on the fragmentation pattern of a particular data structure, and does not necessarily have anything to do with any physical communication network. Only in programs with a very simple structure will it be possible to arrange things so that the same processors will be *neighbors* with respect to all fragmented variables.

Several global communications patterns are useful for operating on variables of different parallel storage classes, and in particular, for transforming data from one storage class to another. Many of these are discussed by Lubeck and Faber[5] in connection with a particle-in-cell program. The simplest of these communications is the broadcast. It carries a *unique* variable in some processor into a *cooperative update* variable, with representatives in all processors. The implementation of broadcast using a logarithmic communication network is often by means of a spanning tree[6]. A variation on the broadcast is to start with a *private* variable, having different values in each processor, and a *fragmented* selection vector, having one logical element in each processor,one of which is *true* and the rest *false*. The broadcast then makes the values of the *private* variable in all processors equal to that in the selected processor. It thus carries a *private* into a *cooperative update* variable with the aid of the selection vector. Note that the *private* variable could be replaced by a *fragmented* vector with one component in each processor.

Another form of the broadcast is the universal broadcast. The adjective, universal, indicates that the same communication is done simultaneously by all processors. In this case, each processor broadcasts a different value to all the rest. A universal broadcast carries either a *private* variable or a *fragmented* structure into a *cooperative update* structure the number of whose components is equal to the number of processors. Both simple broadcast from a *unique* variable and universal broadcast multiply the amount of storage needed by the number of processes.

A communication pattern which is often used to combine partial results computed in separate processors is the generalized accumulate. Although it is called accumulate, it can be done not only with addition but with any associative and commutative operation, such as *multiply*, max, min, and, *or*, *index_of_max*, etc. It carries a *private* variable or *fragmented* vector with one component per processor into a *unique* scalar. It can be carried out by using a spanning tree communication pattern in the direction opposite to that used to broadcast. The universal accumulate provides for the accumulated result to be communicated to all processors. It carries a *fragmented* vector or *private* scalar into a *cooperative update* scalar. The storage required by the

result of an accumulate is less than needed by its operands while universal accumulate does not change the storage needs.

A final important global communication pattern is a universal exchange. In this communication, each processor has a specific item or set of items to send to each other processor. One can think of a two dimensional array of distinct messages indexed by source processor number and destination processor number. The communication starts with a processor holding all messages having its processor number as the source index, and finishes with that processor holding messages having its number as the destination index. The universal exchange carries a *fragmented* structure into a *fragmented* structure having a different mapping. A common use of this communication pattern is to transpose the fragmentation pattern of a matrix, that is, to change a matrix distributed by rows over the processors to one distributed by columns.

The global communications patterns, broadcast, universal broadcast, generalized accumulate, universal accumulate, and universal exchange, all involve considerable communications. On machines with a highly concurrent communications network, however, they all have fairly efficient implementations. If the number of processors is a power of two, they can all be implemented using the butterfly communication pattern with k steps, where k is the base two log of the number of processors. At the jth step, each processor communicates with its 2^jth neighbor. Figure 3 illustrates the idea by
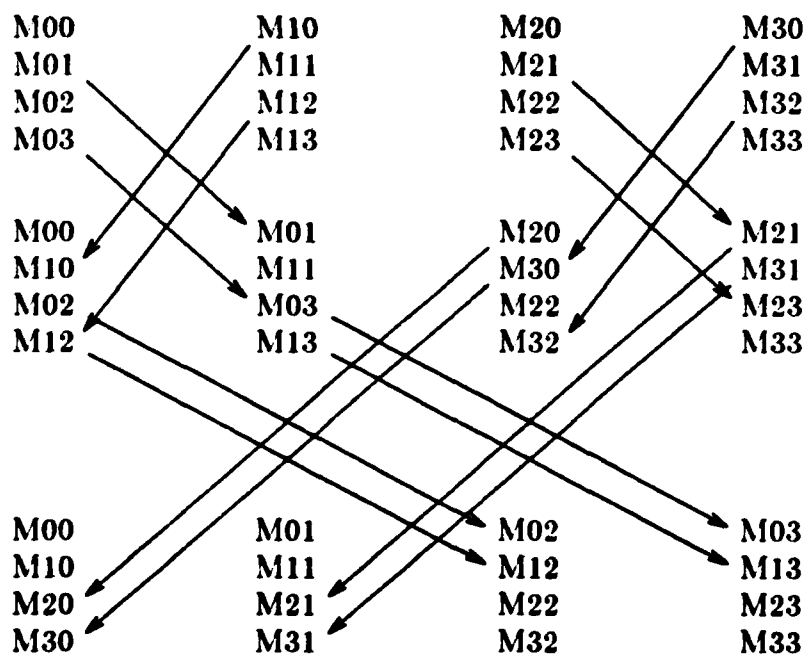

Figure 3: Communication Pattern for Universal Exchange

showing the communications in a four processor system doing universal exchange. It can be seen that neighbors exchange half of their sets of messages at each step. Universal broadcast uses the same communication pattern, but each processor begins with one item to be broadcast, and doubles the size of its result set at each step by including all the results from its neighbor at that step. The same pattern can also be used for universal accumulate, where neighbors exchange single values and then perform the associative operation on the pair to give a single result.

Implications of the Distributed Memory Environment

It can be seen from the above discussion that, even with the most basic model of a distributed memory multiprocessor, in which the topology of the communications network is completely suppressed by assuming a fixed transmission time from any processor to any other, data access limitations are still imposed by the need to fragment large data structures over the processors. The question of how to partition the various *fragmented* data structures of a program is the most basic form of the so-called mapping problem, which has usually been investigated in connection with specific machine topologies [7], [8]. A language for this environment should support the specification of fragmentation patterns, access to *cooperative update* and *replicated* shared variables, and at a minimum, verify accessibility of items in a *fragmented* structure to the processors using them. At the best, we might imagine an automatic assignment of work on *fragmented* structures to the processor having access to the required items.

Some of the implications of distributed memory are best discussed in connection with specific applications. Gaussian elimination without pivoting works well when the matrix is fragmented either by rows or by columns, provided that the loops over row and column indices are nested accordingly, but it is well known that a poor work balance among the processors results if blocks of adjacent rows (or columns) are assigned to the same processor [9]. Gaussian elimination with partial pivoting is somewhat more complicated. In order to allow the search for the pivot element to be done in parallel, the matrix should be fragmented by rows. The problem of singling out the processor containing the current pivot row to do certain operations is a special case of the more general dynamic selection problem. It is often easy to compute a *fragmented* selection vector so that the selected processor has a *true* value for its component while the rest have a *false*, but in order for all processors to know the identity of the selected processor, a broadcast is required. This arises not only in the elimination phase, where a maximum search does the selection, but also in the back substitution if the pivot vector is maintained as a *fragmented* structure.

One very general aspect of the dynamic selection problem deserves mention. In shared memory systems, the technique of self scheduling is often used to allocate work to processors. In this distributed computation, each process can independently determine the next piece of work it is to do, and report that the work has been assigned by a synchronized access to shared

11

memory. In a distributed memory system, it is not possible for a processor to both select a piece of work and to report that selection without communicating with all other processors. Thus the adjective, self, cannot be used to describe the scheduling process, and in this sense, self scheduling is impossible in a distributed memory system.

The particle-in-cell application discussed for distributed memory in [5], and earlier by Hiramoto[10] for shared memory, is a particularly good source of implications. The problem involves two very large data structures, to which parallel operations should be applied, and which need to be *fragmented* for this reason. One is a list of particle positions and velocities, and the other is data associated with a grid of points in Euclidean space, over which a partial differential equation must be solved. Domain decomposition is the appropriate fragmentation pattern for data associated with the grid, but the particles do not remain fixed with respect to the grid, nor is their density uniform over the space. Thus, for load balancing reasons alone, it is not reasonable to fragment the particle data so that each processor has the particles corresponding to its subregion of the grid. Lubeck's solution to this problem was to use some of the communication patterns described above to convert a *fragmented* structure to a shared one at key points in the algorithm.

Conclusions

The Force parallel programming language has shown that it is possible to support a shared memory multiprocessor model of computation which suppresses all explicit process management and requires no topological considerations in order to implement a parallel algorithm in terms of the model. In a distributed memory multiprocessor, there are ways of modeling shared variables and updating them using global communications. In this sense, it is possible to implement a language such as the Force in distributed memory; but it would amount to simulating a shared memory system with a distributed one. The performance of such a simulation is not likely to be acceptable.

The idea of *fragmented* structures seems key to efficient programs for a distributed memory multiprocessor. Topological considerations are introduced into the computation as soon as fragmentation patterns are specified. Even the weakly specified pattern, divide equally over the processors, can lead to difficulties if used on two interacting, but incommensurate, data structures, such as particles and cells. Many problems can be addressed by judicious use of a set of global communications, which should be efficiently supported by the hardware and software of the run time system. The author believes that any general purpose language for large scale, distributed memory multiprocessors will have to be based on extensive support for access to *fragmented* variables and on a careful choice of global communication patterns.

12

REFERENCES

[1] H. F. Jordan, "The Force," in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass, Eds., Chap. 16, MIT Press (1987).

[2] M. E. Conway, "Design of a separable transition-diagram compiler," *Comm. ACM*, Vol. 6, No. 7, 396-408 (1963).

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Comp.*, Vol. C-32, No. 2 (Feb. 1983).

[4] J. S. Kowalik, Ed., *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, MIT Press (1985).

[5] O. M. Lubeck and V. Faber, "Modeling the Performance of Hypercubes: A Case Study Using the Particle-In-Cell Application," Los Alamos National Laboratory document LAUR 87-1522, submitted to *Parallel Computing*, (1987).

[6] S. L. Johnsson and Ching-Tien Ho, "Spanning Graphs for Optimum Broadcast and Personalized Communication in Hypercubes," Yale University report YALEU/DCS/TR-500 (Nov. 1986).

[7] S. Bokhari, "On the Mapping Problem," *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207-214 (March 1981).

[8] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," *Proc. 1984 Int'nl Conf. on Parallel Processing*, (Aug. 1984).

[9] G. A. Geist and M. T. Heath, "Matrix Factorization on a Hypercube Multiprocessor," in *Hypercube Multiprocessors 1986*, pp. 161-180, SIAM, Philadelphia (1986).

[10] R. Hiramoto, "Some Issues in Parallel Processing as Encountered on the Denelcor HEP," *Parallel Computing*, Vol. 3, No. 2, pp. 111-127 (May 1986).

BIBLIOGRAPHIC DATA SHEET	1. Report No. ECE Technical Report 87-1-3	2.	3. Recipient's Accession No.

4. Title and Subtitle		5. Report Date September 1987
Programming Language Concepts for Multiprocessors		6.

7. Author(s) Harry F. Jordan	8. Performing Organization Rept. No. CSDG 87-4

9. Performing Organization Name and Address	10. Project/Task/Work Unit No.
Computer Systems Design Group Department of Electrical and Computer Engineering University of Colorado Boulder, CO 80309-0425 USA	11. Contract/Grant No. 9-XSH-5604M-1

12. Sponsoring Organization Name and Address	13. Type of Report & Period Covered Interim
Los Alamos National Laboratory P.O. Box 990 Los Alamos, NM 87545	14.

15. Supplementary Notes

Also supported in part by ONR under Grant N00014-86-k-0204.

16. Abstracts It is now possible to build multiprocessor systems to support the tightly coupled activity of hundreds of thousands of different instruction streams, or processes, either by coupling many monoprocessors, or a smaller number of pipelined multiprocessors, through a high concurrency switching network. This network may couple processors to memory modules, making a shared memory multiprocessor system, or it may couple processor/memory pairs, making a distributed memory system.

The need to direct very many processes simultaneously places qualitatively different demands on a programming language than the direction of a single process. In spite of the different requirements, most languages for multiprocessors have just extended conventional single stream programming languages. The extensions are often implemented by way of subroutine calls and have little impact on the basic structure of the language. This paper attempts to examine the underlying conceptual structure of parallel languages for large scale multiprocessors on the basis of an existing language for share memory multiprocessors, known as the Force, and to extend the concepts in this language to distributed memory systems.

17. Key Words and Document Analysis. 17a. Descriptors

multiprocessor
programming language
shared memory
distributed memory
parallel programming

17b. Identifiers/Open-Ended Terms

the Force

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70)   USCOMM-DC 40329-P71

END

DATE

FILMED

DTIC

JULY 88